AN APPLICATION OF THE FIXPOINT THEORY TO COMPLEXITY OF PROGRAMS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
AJITBHAI M. SUTHAR

to the
COMPUTER SCIENCE PROGRAM
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

MAY 1980

I. I. T. KANPUR
CENTRAL LIBRARY
Acc. No. A 63018
- 8 AUG 1980

CSP-1980-M-SUT-APP

4.6.80

CERTIFICATE

This is to certify that the work entitled "AN APPLICATION OF THE FIXPOINT THEORY TO COMPLEXITY OF PROGRAMS" has been carried out by Sri Ajitbhai Mohanlal Suthar under my supervision and has not been submitted elsewhere for the award of a degree.

[illegible]

Kanpur
28 May 1980

Kesav V. Nori
Assistant Professor
Computer Science Program
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Professor Kesav V. Nori for his invaluable suggestions, constant encouragement and guidance. Without his help I would not have come to know about applications of pure mathematics to computer science.

My sincere thanks to all my classmates and other colleagues who encouraged me during my studies here.

Last but not the least, I thank Mr. H.K. Nathani for his excellent typing.

28th May 1980 — Ajitbhai M. Suthar

CONTENTS

ABSTRACT

In this thesis we make a modest attempt to explore the possibility of using an established method of proving properties of programs, viz., The Fixpoint Theory of Computation, to prove results concerning the Time Complexity of a restricted class of programs written in an ALGOL-like language. This attempt may be viewed as a case for uniformity of study of various properties of programs.

CHAPTER 1

INTRODUCTION

The Fixpoint Theory of Computation of Scott (Scott 70) is used to define the Semantics of Computer Programs in terms of the least fixpoint of recursive transformations of the state (a name-value mappling). This allows not only the justification of all existing verification techniques, but also their extension to the handling in a uniform manner, of various properties of computer programs, including correctness, termination and equivalence (Manna 74).

The study of Computational Complexity of Programs (Aho 74), one of the important properties of programs, is not yet handled through The Fixpoint Theory. It is studied as a separate field. Thus, there is no uniformity of approach in the study of important properties of programs.

A possibility that can be suggested for applying the fixpoint theory to the study of computational complexity arises from the manner in which the function APPLY is defined in LISP (McCarthy 60):

APPLY : LF XAL $\longrightarrow$ SETP

where LF is a LISP function,
AL is a list of arguments,
SETP is the set of values to which LISP functions reduce.

APPLY gives the semantics of LISP functions. The suggestion here is, can we analogously define:

TCOMPLEXITY : $F \times TP^m \times PS^r \longrightarrow G$

SCOMPLEXITY : $F \times SP^n \times PS^r \longrightarrow H$

where

F is a computable function,

PS is a characterization of problem size,

TP is a set of primitive functions in terms of which F is recursively defined and which are considered as significant with respect to the assessment of time complexity,

SP is a set of primitive data units in terms of which we wish to assess space-complexity,

$G : TP^m \times PS^r \longrightarrow R^m$,

$H : SP^n \times PS^r \longrightarrow R^n$

R is the set of real numbers

and TCOMPLEXITY and RCOMPLEXITY are functions that respectively stand for Time and Space complexity of F with respect to PS and the corresponding primitives TP and SP.

The motivation for this suggestion is that we try to throw away all information in the program that is irrelevant as far as the study of computational complexity goes. This done, we would have, as a result a simple function which, in essence, counts. Appreciating the bounds on the value of the counters, one for each primitive function or data unit is tantamount to appreciating the complexity of the original function.

A possible track for effecting the above reduction is to build a library of typical counting functions and proofs of their bounds separately. This activity need be done only once, of course. The library can then be effectively used for obtaining the bounds of the resultant G and H respectively.

The above approach is indeed a long range one. In this thesis, we make a modest attempt, as a beginning, in the above view, for a restricted class of simple programs. For such a class of programs P, we try to formulate the transformation to a class of recursively specified programs P', such that the number of computations needed in P' is identical to that of P. We then apply The Fixpoint Theory to find out the complexity of P'.

1.1 OUTLINE OF THE THESIS

To understand The Fixpoint Theory we will require many important mathematical terms. We survey these briefly in this chapter, in the next section.

Chapter 2 gives a brief introduction to The Fixpoint Theory of Computation. Scott's approach is based on the following central idea:

"The functions computed by Recursive Programs are the least fixpoints of monotonic and continuous transformations that they represent".

Scott's construction of the least fixpoint of a program T as the least upper bound of the sequence of functions $T^i(w)$ is considered in this chapter.

In Chapter 3 it is shown how high level language (ALGOL-like) programs can be translated in to recursive programs of the form:

$$F(\bar{x}) \Leftarrow T(F)(\bar{x}).$$

Chapter 4 contains an application of The Fixpoint Theory, to a restricted class of programs, to find their complexity.

1.2 BASICS

We shall consider n-ary partial functions ($n \geq 0$, integer) from a domain $D^n = D_1 X D_2 X \dots X D_n$ into range D'; that is, for every n-typle $\bar{a} = (a_1, a_2, \dots, a_n) \in D^n$, either $f(\bar{a})$ is defined (i.e., $f(\bar{a}) \in D'$) or it is undefined.

A 0-ary partial function is a function with n arguments yielding either a constant value or is "undefined".

We shall include n-ary partial predicates $p(\bar{x})$ as a special case because they are partial functions from D^n to $D' = \{true, false\}$.

In developing a theory for handling partial functions it is convenient to introduce the special element w to represent the value "undefined". Thus if $f(\bar{a})$ is undefined, we write $f(\bar{a}) = w$.

Since we shall consider composition of functions, we may need to compute functions with some arguments being w.

Let us take $D^n_+ = D^n \cup \{w\}$ and

$$D'_+ = D' \cup \{w\}$$

then partial functions from D^n to D' are total functions from D^n_+ to D'_+. So extension of functions from D^n into D' to D^n_+ into D'_+ assures the possibility of composition.

We will use w to stand for

(a) the single undefined element w

(b) the undefined vector $(w, w, \dots, w)$

or (c) function w which is undefined for all domain elements.

Depending upon the context the appropriate use of w will be unambiguously understood (Manna 74).

Definition 1.2.1 COMPOSITION

Let $f : D_+^n \longrightarrow D_+'$ and $g : D_+' \longrightarrow D_+''$ $(D_+'' = D'' \cup \{w\})$ then composite of g and f, denoted by $g_\circ f$, is defined by

$$g_\circ f(\bar{x}) = g(f(\bar{x})); \quad \forall \ \bar{x} \in D_+^n .$$

(For the composition $g_\circ f$ "domain of g equal to range of f" is not necessary but the range of f should be the subset of domain of g.)

Definition 1.2.2 RULE OF COMPUTATION

Given a composite function $g_\circ f$, the order of evaluation is called a rule of computation. For instance $g_\circ f(\bar{x})$ can be evaluated by the following two sequences:

Let $f(\bar{x}) = \bar{y}$ then

one sequence is (1) $\bar{y} = f(\bar{x})$

then (2) $g(\bar{y})$.

Another sequence is obtained by substituting $f(\bar{x})$ for $\bar{y}$ throughout the specification of g.

In the first sequence, the computation of $f(\bar{x})$ is done only once, whereas in the second sequence, $f(\bar{x})$ is computed wherever the value of $\bar{y}$ is required in the specification of g.

Definition 1.2.3 PARTIALLY ORDERED SET (poset)

A poset is a set D in which a binary relation $\sqsubseteq$ is defined, which satisfies for all $x,y,z \in D$ the following conditions:

(a) $x \sqsubseteq x$ for all $x \in D$ (Reflexive)

(b) If $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$ (Antisymmetry)

(c) If $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$ (Transitivity)

Definition 1.2.4 CHAIN

A poset D with respect to $\sqsubseteq$ is said to be a chain (or totally ordered or linearly ordered) if for all $x,y \in D$ either $x \sqsubseteq y$ or $y \sqsubseteq x$. i.e., all elements of D are comparable.

Definition 1.2.5 A CHAIN IN POSET

Let D be a poset, K a non-empty subset of D and $\sqsubseteq'$ the restriction of $\sqsubseteq$ to K,

$$\sqsubseteq' = \sqsubseteq \cap (K \times K)$$

i.e., for $x,y \in K$, $x \sqsubseteq' y$ if and only if $x \sqsubseteq y$.

If K with respect to $\sqsubseteq'$ is a chain, we say that K is a chain in D.

Definition 1.2.6 BOUNDS

Let D be a poset and $K \subseteq D$. An element $d \in D$ is said to be an upper bound (u.b) of K if for every $x \in K$, $x \sqsubseteq d$.

Note that a subset $K \subseteq D$ may have more than one u.b. in D or may have no u.b. in D.

An element $l \in D$ is said to be the least upper bound (lub) of $K \subseteq D$ if it is the least element of all upper bounds of K in D. That is, it satisfies

(a) for all $x \in K$, $x \sqsubseteq l$

and (b) for any u.b. $d \in D$ of K, $l \sqsubseteq d$

This lub is denoted by $\bigsqcup K$.

If $K = \{x_1, x_2, \ldots, x_n\}$ we shall write

$$K = \bigsqcup_{i=1}^{n} x_i \text{ and if } K = \{x_1, x_2, \ldots, x_n, \ldots\}$$

then

$$K = \bigsqcup_{i=1}^{\infty} x_i .$$

Definition 1.2.7 A CHAIN-CLOSED SET

A pos et D is said to be chain-closed if each chain in D has a lub.

It is very easy to verify that

(a) a subset of a chain is itself a chain.

(b) each finite chain has a lub.

Definition 1.2.8

Let f be a function with a domain D and $x \in D$. For any $n \in N$ (the set of non-negative integers), we define the notation $f^n(x)$ as follows:

(a) $f^{\circ}(x) = x$

(b) $f^{n+1}(x) = f(f^n(x))$; $n \geqslant 0$.

A fixpoint of f is an element $x \in D$ for which $f(x) = x$.

When the set of all fixpoints of f has a least element, this element is called the least fixpoint of f.

Definition 1.2.9 MONOTONICITY

Let D_1 and D_2 be two posets with $\sqsubseteq$ and $\sqsubseteq'$ as relations respectively. Then a function $f : D_1 \longrightarrow D_2$ is said to be monotonic if for all $x,y \in D_1$,

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq' f(y).$$

Now let us take an example which will help in understanding all these terms. In the following chapters we will use the same order relation over the set of functions and notations which are given in the example below:

<u>Example</u>: Let S be the set of all partial functions from D' to D''.

Define $\sqsubseteq$ on S as follows:

For any $f, g \in S$,

$$f \sqsubseteq g \text{ if either } f(x) = w \text{ or}$$
$$f(x) = y \text{ then } g(x) = y; \; (y \neq w)$$

i.e., whenever f is defined at x, g is also defined at x and they have the same value at x.

So note that $w \sqsubseteq f$ for any $f \in S$.

One can easily verify that S is a poset.

Let T be a function from $S \longrightarrow S$ (i.e., T is a functional).

$\therefore$ T is monotonic if and only if

$$T(f) \sqsubseteq T(g) \text{ whenever } f \sqsubseteq g.$$

If T is a monotonic functional then since

$w \sqsubseteq T^1(w)$, $T^\circ(w) \sqsubseteq T^1(w)$. $\quad (\because T^\circ(w) = w)$

$\therefore T^1(w) = T(T^\circ(w)) \sqsubseteq T(T^1(w)) \quad (\because T \text{ is monotonic})$

$= T^2(w).$

Thus $T^1(w) \sqsubseteq T^2(w)$. By applying T to this we get $T^2(w) \sqsubseteq T^3(w)$ and so on.

$\therefore$ We have

$$T^o(w) \sqsubseteq T^1(w) \sqsubseteq T^2(w) \sqsubseteq \;\ldots\; \sqsubseteq T^n(w) \sqsubseteq T^{n+1}(w) \sqsubseteq \;\ldots$$

Thus

$$\{T^i(w) \;:\; i \geqslant 0 \text{ integer}\} \text{ is a chain in S.}$$

$\therefore$ If S is a chain-closed set, it has a lub $\bigsqcup_{i=0}^{\infty} T^i(w)$ in S (Loeckx 74). **

CHAPTER 2

THE FIXPOINT THEORY

2.1 FUNDAMENTAL VIEW OF PROGRAMS

A program P describes an action, usually called a command or statement. As we look within this program we find this gross command to be a composition of simpler commands.

What is a command? A command changes the state of the store (memory). That is,

a command (statement) is a mapping of initial state to a final state.

A State S of the store can be described by the contents of locations, i.e., it gives values of the locations. So we can view a state as a function from names (locations) to values.

Thus, if L is the set of locations and

V is the set of values then

$S : L \longrightarrow V$.

One may think of the state as consisting of all variables (names manipulated by the program) and their respective values.

Thus we see that a statement is a nothing but a function of function i.e., a functional.

In other words,

A Computer Program is a functional.

2.2 RECURSIVE PROGRAMS

A Recursive Program is a lisp-like definition of the form

$$f(\bar{x}) \Leftarrow T(f)(\bar{x})$$

where T is a functional, T(F) is a function.

T(F) is a composition of base functions (like assignment, addition, subtraction etc.) and the function variable F, applied to the individual variables $\bar{x} = (x,y,z, \ldots)$.

For example, P_0, shown below, is a recursive program over the integers.

P_0 : $F(x) \Leftarrow$ If $x = 0$ then 1 else $F(x-1)$.

Note that it is undefined for negative integers (does not terminate for negative integers), so it is a partial function.

We allow our base functions to be partial because they represent the result of some computation which may, in general, give results for some inputs and run indefinitely for others.

As limiting cases of partial functions we include the nowhere defined function w and everywhere defined functions, called total functions.

Example: The statement

L : GO TO L has undefined effect for all state vectors . **

For any $n \geq 0$ and given domains D' and D'' (where D' may be $D_1 \times D_2 \times \ldots \times D_n$), let us denote the set of all functions

$f : D'_+ \longrightarrow D''_+$ and predicates over D'

as 'Fun'. Fun includes nowhere defined function w also.

In the sequel n is an integer, $n \geq 0$; F or f denotes a function variable ranging over Fun, i.e., $F, f \in$ Fun; $\bar{x}$ denotes a variable ranging over D'; T is a function, T : Fun $\longrightarrow$ Fun.

On every extended domain $D_+ = D \cup \{w\}$ we define the partial ordering in the following way (Manna 74):

for $x, y \in D_+$,

$$x \sqsubseteq y \text{ if and only if either } x = w \text{ or } x = y.$$

Note that distinct elements of D_+ are unrelated by $\sqsubseteq$ and $w \sqsubseteq x$ for all $x \in D_+$.

For $(D_+)^n = D_+ \times D_+ \times \dots \times D_+$ we define

$$(x_1, x_2, \dots, x_n) \sqsubseteq (y_1, y_2, \dots, y_n)$$

if and only if $x_i \sqsubseteq y_i$ for each i, $1 \leq i \leq n$.

If f is unary function from D'_+ to D''_+ we define $f(w) = w$.

If f is n-ary $(n \geq 2)$ function from D'_+ to D''_+ we define

$$f(x_1, x_2, \dots, x_n) = w \text{ if at least one } x_i \text{ is } w.$$

Thus all our basic functions (i.e. members of <u>Fun</u>) are monotonic.

2.3 ORDERING

The aim of this chapter is to prove that if F is recursively defined by T(F) then F is the lub of $\{T^i(w) : i \geq 0, \text{integer}\}$, i.e., $F(\bar{x}) \Leftarrow T(F)(\bar{x})$ then $F = \bigsqcup_{i=0}^{\infty} T^{i}(w)$.

A key question to this end is: To which order relation, does this lub correspond?

Let us define the partial ordering (relation) $\sqsubseteq$ on Fun, the set of functions as below:

Definition 2.3.1

Let $f, g \in$ Fun then $f \sqsubseteq g$ if and only if either $f(\bar{x}) = w$

or $f(\bar{x}) = y \Rightarrow g(\bar{x}) = y$

where $y \neq w$.

i.e., wherever f is defined, g is also defined and both give the same value for that argument.

Examples

(a) For any function $f \in$ Fun, $w \sqsubseteq f$.

(b) Let $D' = I \times I$ and $D'' = I$ (I is the set of all integers).

Define $f_1, f_2 \in$ Fun as follows:

$f_1(x,y) =$ If $x \geqslant y$ then $x + 1$ else $y - 1$

$f_2(x,y) =$ If $(x \geqslant y)$ and $(x-y$ even$)$ then $x + 1$ else w.

We see that for any pair of integers (x,y) such that $f_2(x,y)$ is defined, $f_1(x,y)$ is also defined and have the same value, $x + 1$.

Thus $f_2 \sqsubseteq f_1$. **

Definition 2.3.2 Syntax for T(F)

The syntax of a functional T is defined as follows:

T(F) is either

(i) a base function

(ii) F

(iii) constructed from already given $T_1(F)$, $T(F)$ by

(a) composition : $T_1(F) \circ T_2(F)$ or

(b) selection :

<u>If</u> $P(\bar{x})$ <u>then</u> $T_1(F)(\bar{x})$ <u>else</u> $T_2(F)(\bar{x})$

or (iv) Constructed from already given

$T_1(F), \ldots, T_m(F)$ $(m \geqslant 1)$ and given m-variable base function b giving

$$b \circ (T_1(F), \ldots, T_m(F)).$$

E.g., $F(x) \Leftarrow$ <u>If</u> $x = 0$ <u>then</u> 1 <u>else</u> $x * F(x-1)$ defined over integers.

We can see that $T(F)$ is constructed from many functions : $f(x) = 1$; $g(x) = x-1$; base function $b(x,y) = x * y$; predicate $p(x) = (x = 0)$, all according to the syntax given in the definition.

<u>Theorem 2.1</u>

Let $T(F)$ be given as in Definition 2.3.2 then T is a monotonic functional, i.e., for any F_1, F_2 such that $F_1 \sqsubseteq F_2$, $T(F_1) \sqsubseteq T(F_2)$.

<u>Proof</u>

Let us prove it by induction on the structure of T.

(i) If $T(F)$ = base function f

i.e., $T(F)\ (\bar{x}) = \bar{y}$ if $f(\bar{x}) = \bar{y}$

or $T(F) = F$ then it is obvious.

(ii) Let $T(F) = T_1(F) \circ T_2(F)$ where T_1 and T_2 are monotonic. (Induction hypothesis).

Let $F_1 \sqsubseteq F_2$ and $T(F_1)(\bar{x}) = \bar{y}$ then by definition of composition, there exists $\bar{z}$ such that

$$T_2(F_1)(\bar{x}) = \bar{z} \text{ and } T_1(F_1)(\bar{z}) = \bar{y}.$$

Now $F_1 \sqsubseteq F_2$ so by induction hypothesis,

$$T_1(F_1) \sqsubseteq T_1(F_2) \text{ and } T_2(F_1) \sqsubseteq T_2(F_2)$$

$\therefore$ $T_2(F_1)(\bar{x}) = \bar{z} \implies T_2(F_2)(\bar{x}) = \bar{z}$

and $T_1(F_1)(\bar{z}) = \bar{y} \implies T_1(F_2)(\bar{z}) = \bar{y}.$

$\therefore$ $T_1(F_2) \circ T_2(F_2)(\bar{x}) = \bar{y}.$

$\therefore$ $T(F_2) = \bar{y}$

Thus $T(F_1)(\bar{x}) = \bar{y} \implies T(F_2)(\bar{x}) = \bar{y}$

$\therefore$ $T(F_1) \sqsubseteq T(F_2)$

$\therefore$ T is monotonic.

The remaining cases can be proved similarly. **

2.4 THE FIXPOINT THEOREM

Now let us see how the least fixpoint of a transformation T is obtained.

We know that $w \sqsubseteq f$ for any function f in Fun. Now consider the sequence

$$w,\ T^1(w),\ T^2(w),\ \ldots,\ T^i(w),\ \ldots$$

Since T is monotonic,

$w \sqsubseteq T^1(w)$ i.e., $T^0(w) \sqsubseteq T^1(w)$ $\therefore$ $T(T^0(w)) \sqsubseteq T(T^1(w))$

$\therefore$ $T^1(w) \sqsubseteq T^2(w)$

Again applying T, we get $T^2(w) \sqsubseteq T^3(w)$ and so on.

So $w \sqsubseteq T^1(w) \sqsubseteq \ldots\ldots \sqsubseteq T^i(w) \sqsubseteq \ldots$

$\therefore \{T^i(w) : i \geqslant 0 \text{ integer}\}$ is a chain in Fun.

But Fun is a chain-closed set (see(Loeckx 74), or (Manna 74))

So $\{T^i(w) : i \geqslant 0 \text{ integer}\}$ has a lub $\bigsqcup_{i=0}^{\infty} T^i(w)$.

In this section we shall prove that $\bigsqcup_{i=0}^{\infty} T^i(w)$ is a least fixpoint of a transformation T.

But for that we need to prove that T is a continuous function.

Definition 2.4.1 CONTINUITY

Suppose that M is a chain-closed set. The function f defined on M is said to be continuous if for any chain $K \subseteq M$

(i) $f(K)$ has a lub

(ii) $f(\bigsqcup K) = \bigsqcup f(K)$.

Note that, by rearranging the elements, if necessary, in any chain $K = \{f_i\}$ of, functions in Fun, K can be rewritten as $\{f_i\}$ such that $f_j \sqsubseteq f_{j+1}$ for every $j \geqslant 0$. Also T is monotonic, so we can see that $T(K)$ is also a chain. And Fun being a chain-closed set, $T(k)$ has a lub. So to prove the continuity of T we have to prove that for any chain of functions $\{f_i\}$, $T(\bigsqcup\{f_i\}) = \bigsqcup\{T(f_i)\}$.

Theorem 2.2

Any functional T defined as in Definition 2.3.2 is continuous.

Proof

We prove the theorem by induction on the structure of T.

The details of the proof are given in (Bakker 75).

Theorem 2.3

Let F be declared by $F(\bar{x}) \Leftarrow T(F)(\bar{x})$ where T is defined as in Definition 2.3.2, then $\bigsqcup_{i=0}^{\infty} T^i(w)$ is the least fixpoint of T.

Proof

Since T is monotonic function, $\{T^i(w)\}_{i=0}^{\infty}$ is a chain in Fun. So $\bigsqcup_{i=0}^{\infty} T^i(w)$ exists.

Let

$$\bigsqcup_{i=0}^{\infty} T^i(w) = F_T$$

(i) F_T is a fixpoint of T

$$T(F_T) = T\left(\bigsqcup_{i=0}^{\infty} T^i(w)\right)$$

$$= \bigsqcup_{i=0}^{\infty} T(T^i(w)) \quad (\because \text{ T is continuous})$$

$$= \bigsqcup_{i=0}^{\infty} T^{i+1}(w) = \bigsqcup_{j=1}^{\infty} T^j(w) = \left(\bigsqcup_{j=1}^{\infty} T^j(w)\right) \sqcup w$$

$$= \bigsqcup_{j=0}^{\infty} T^j(w)$$

$$= F_T$$

(ii) F_T is the least fixpoint.

Suppose that a function f is any fixpoint of T. i.e., $T(f) = f$.

We have to prove that $F_T \sqsubseteq f$.

By induction on i, we prove that $T^i(w) \sqsubseteq f$ for any non-negative integer i.

For $i = 0$, $T^i(w) = T^0(w) = w$ $\therefore w \sqsubseteq f \Rightarrow T^0(w) \sqsubseteq f$.

Suppose for $i = j$, $T^i(w) \sqsubseteq f$ holds. $\therefore T^j(w) \sqsubseteq f$

Now

$$T^{j+1}(w) = T(T^j(w))$$

$$\sqsubseteq T(f) \qquad (\because T^j(w) \sqsubseteq f \text{ and } T \text{ is monotonic})$$

$$= f$$

$$\therefore T^{j+1}(w) \sqsubseteq f$$

Thus $T^j(w) \sqsubseteq f \Rightarrow T^{j+1}(w) \sqsubseteq f$

$\therefore$ $T^i(w) \sqsubseteq f$ for every integer $i \geqslant 0$.

$\therefore$ f is an upper bound of $\{T^i(w)\}_{i=0}^{\infty}$.

$$\therefore \bigsqcup_{i=0}^{\infty} T^i(w) \sqsubseteq f \tag{2.4.1}$$

Thus for any fixpoint f, $F_T \sqsubseteq f$.

$\therefore$ F_T is the least fixpoint of T.

2.5 SEMANTICS OF A RECURSIVE PROGRAM

In this section we shall show that if a function F is recursively defined by

$$F(\bar{x}) \Leftarrow T(F)(\bar{x})$$

then F can be found by the least fixpoint of T.

In fact,

$$F = \bigsqcup_{i=0}^{\infty} T^i(w)$$

Theorem 2.4.

Let $F(\bar{x}) \Leftarrow T(F)(\bar{x})$ then $F = \bigsqcup_{i=0}^{\infty} T^i(w)$.

(Syntax of T is as in Definition 2.3.2).

Proof

We have to justify (i) $\bigsqcup_{i=0}^{\infty} T^i(w) \sqsubseteq F$

and (ii) $F \sqsubseteq \bigsqcup_{i=0}^{\infty} T^i(w)$

(i) $F(\bar{x}) \Leftarrow T(F)(\bar{x})$ So F is a fixed point of T (Bakker 75▲).

So from the result (2.4.1) we have

$$\bigsqcup_{i=0}^{\infty} T^i(w) \sqsubseteq F.$$

(ii) We have to prove that if $F(\bar{x})$ is defined, say $F(\bar{x}) = \bar{y}$ then

$$(\bigsqcup_{i=0}^{\infty} T^i(w))(\bar{x}) = \bar{y}$$

$F(\bar{x})$ is defined. How $F(\bar{x})$ is determined? Let us look at the way in which the programmer determines $F(\bar{x})$.

He/She determines $T(F)(\bar{x})$. For this, he/she may call F again i.e., T(F) is substituted in place of F.

Since $F(\bar{x})$ is defined, this calling process must come to an end. i.e., there exists an $j \geq 1$ such that $T^j(F)(\bar{x}) = \bar{y}$ and in the application of $T^j(F)$ to the given $\bar{x}$, F is not encountered anymore.

And so, we may replace F in $T^j(F)$ everywhere by w.

i.e., For this $\bar{x}$, we can write

$$\bar{y} = T^j(F)(\bar{x}) = T^j(w)(\bar{x})$$

$$\therefore \quad T^i(w)(\bar{x}) = \bar{y} \quad \text{for every } i \geq j \text{ and } j \text{ is finite.}$$

Now $\bigsqcup_{i=0}^{\infty} T^i(w)$ is the limit of the sequence $\{T^i(w)\}$ (Scott 70), so above result implies

$$\left(\bigsqcup_{i=0}^{\infty} T^i(w) \right)(\bar{x}) = \bar{y}$$

$$\therefore \quad F \sqsubseteq \bigsqcup_{i=0}^{\infty} T^i(w) \qquad **$$

Thus the semantics of a recursive program can be defined as:

"The function defined by a recursive program $F \Leftarrow T(F)$ is the least fixpoint of T, viz., $\bigsqcup_{i=0}^{\infty} T^i(w)$."

2.6 Example

Consider the recursive program over the set of non-negative integers.

$F(x) \Leftarrow$ If $x = 0$ then 1 else $x * F(x-1)$.

Here $T(F)(x) =$ If $x = 0$ then 1 else $x * F(x-1)$.

$T^0(w) = w$.

$T^1(w)(x) =$ If $x = 0$ then 1 else $x * w(x-1)$

$=$ If $x = 0$ then 1 else w.

Thus $T^1(w)$ is defined only for $x = 0$ and its value for $x = 0$ is 1.

$T^2(w)(x) = (T(T^1(w)))(x)$

$= T(T^1(w)(x))$

$=$ If $x = 0$ then 1 else $x * T^1(w)(x-1)$

$=$ If $x = 0$ then 1 else $x *$ (If $x-1 = 0$ then 1 else w)

Thus $T^2(w)$ is defined only for $x = 0,1$.

And $T^2(w)(0) = 1$, $T^2(w)(1) = 1$.

$T^3(w)(x) =$ If $x = 0$ then 1 else

$x *$ (If $x = 1$ then 1 else

$(x-1) *$ (If $x = 2$ then 1

else w))

For $x > 2$, $T^3(w)$ is undefined.

And $T^3(w)(0) = T^3(w)(1) = 1$, $T^3(w)(2) = 2 * (2-1) * 1$

$$= 2$$

In general,

$$T^i(w)(x) = \underline{\text{If}}\ x \geqslant 0\ \underline{\text{then}}\ x\,!\ \ \underline{\text{else}}\ w.$$

$$\therefore \bigsqcup_{i=0}^{\infty} T^i(w) = F_T \text{ where } F_T(x) = x\,! \text{ for every integer } x \geqslant 0.$$

**

To avoid confusion with a recursive definition of F we shall denote $\bigsqcup\limits_{i=0} T^i(w)$ by F_T instead of F, in the following chapters.

I I [illegible] PUR
CENTRAL LIBRARY
Acc. No. A 63018

CHAPTER 3

TRANSFORMATION OF ALGOL-LIKE PROGRAMS TO RECURSIVE PROGRAMS

The scope of this chapter is the specification of transformation rules that will enable the rewriting of programs written in ALGOL-like languages to Recursive Programs as described in Section 2.2. The purpose of this transformation is to be able to apply the theory available for studying properties of Recursive Programs to the study of time complexity of simple programs written in practical languages which is the subject matter of Chapter 4.

The sole guiding principle for the transformation of ALGOL-like programs P to Recursive Programs P' is that (Manna 72)

(i) the partial functions computed respectively by both programs are identical; and

(ii) the transformation preserves the time complexity properties of P.

The features of ALGOL that we consider are very simple indeed; there is, however, no theoretical difficulty in extending them. The language we consider has: (E is an expression; B, B_1, B_2 are blocks or elementary statements)

(1) Assignment statements: $x_i := E(\bar{x})$ where $\bar{x} = (x_1, \ldots, x_i, \ldots, x_n)$

(2) Sequential composition: $B_1 : B_2$

(3) Conditional composition:

(i) <u>If</u> $p(\bar{x})$ <u>then</u> B

(ii) <u>If</u> $p(\bar{x})$ <u>then</u> B_1 <u>else</u> B_2

(4) Iterative composition: <u>while</u> $p(\bar{x})$ <u>do</u> B

(5) Procedural abstraction:

(i) non recursive procedures:

<u>Procedure</u> P; B

(ii) recursive procedures:

<u>Procedure</u> P; B[P]

As a major restriction, whose impact will be seen only in Chapter 4, we consider only those programs in which iterative statements as well as control of recursion are effected by Boolean conditions concerning a relationship between a single variable (which is modified within the enclosed block) and other constants (that is, either literals or variables which are not modified by any constitutent command within the enclosed block).

3.1 RULES OF TRANSFORMATION

The rules of transformation, described in the following subsections, are based on the description of the ALGOL-like subset above. The transformation is defined blockwise: to each block B (or elementary statement) we associate a partial function f_B computed by it.

For example:

<u>begin</u> x:= x-2; y:= y+1; z:=z-1 <u>end</u>,

will be rèpresented by the function

$$f(x,y,z) = (x-2,\ y+1,\ z-1).$$

3.1.1 Assignment Statements

If B is x:= $E(\bar{x})$ then

$$f_B(\bar{x}) = (x_1,\ x_2,\ \ldots,\ x_{i-1},\ E(\bar{x}),\ x_{i+1},\ \ldots,\ x_n)$$

where

$\bar{x} = (x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$

Note, as a result, that side effects in expressions are disallowed.

3.1.2 Sequential Composition

If $B = B_1; B_2$ then $f_B(\bar{x}) = f_{B_2}(f_{B_1}(\bar{x}))$. Under the rule of computation that B_1 is performed before B_2, we assert that,

$$f_B(\bar{x}) = f_{B_2}(\bar{y}) \text{ where } \bar{y} = f_{B_1}(\bar{x})$$

3.1.3 Conditional Composition

3.1.3.1 For the case $B = \underline{\text{If}}\ p(\bar{x})\ \underline{\text{then}}\ B_1;$

$$f_B(\bar{x}) = \text{If } p(\bar{x}) \text{ then } f_{B_1}(\bar{x}) \text{ else } \bar{x}.$$

Note that the Boolean condition p on $\bar{x}$ is side-effect free.

3.1.3.2 For the second case

$$B = \underline{\text{If}}\ p(\bar{x})\ \underline{\text{then}}\ B_1\ \underline{\text{else}}\ B_2;$$

$$f_B(\bar{x}) = \text{If } p(\bar{x}) \text{ then } f_{B_1}(\bar{x}) \text{ else } f_{B_2}(\bar{x}).$$

3.1.4 Iterative Composition

$B = \underline{\text{while}}\ p(\bar{x})\ \underline{\text{do}}\ B_1$

then $f_B(\bar{x})$ is the least fixpoint of the recursive program

$$F(\bar{x}) \Leftarrow \underline{\text{If}}\ p(\bar{x})\ \underline{\text{then}}\ F(f_{B_1}(\bar{x}))\ \underline{\text{else}}\ \bar{x}.$$

3.1.5 Procedures

Here we have two cases : non-recursive procedures and recursive procedures. We will deal with each in turn.

3.1.5.1 Non-recursive Procedures: Consider the declaration

Procedure P; B_1

where P is the name of the procedure; and

B is its body. Intuitively, we see that

B_1 is executed on call. This is what we wish to formalize:

B = Call P then

$f_B = f_{Call\ P}(\bar{x}) = f_{B_1}(\bar{x})$

3.1.5.2 Recursive Procedures: Consider now the declaration

Procedure P; $B_1[P]$

where P is the name of the procedure; and

$B_1[P]$ is a block in which there is a recursive call to P.

$f_{Call\ P}(\bar{x})$ is the least fixpoint of the recursive program

$$F(\bar{x}) \Leftarrow f_{B_1[P]}(\bar{x})$$

Where occurrences of Call P will be replaced by F in the sematic definition $f_{B_1[P]}$.

Example:

Consider the program P which computes x!

for given non-negative integer x.

```
P : begin y:=1;
          while x > 0 do
                    begin y:= y*x;
                          x:=x-1
                    end
    end.
```

The partial function computed by P is identical to the least fixpoint of p' where

$P' : G(x) \Leftarrow F(x,1)$

$F(x,y) \Leftarrow$ If $x > 0$ then $F(x-1,\ y*x)$ else (x,y).

CHAPTER 4

APPLICATION OF THE FIXPOINT THEORY TO STUDY TIME-COMPLEXITY OF PROGRAMS

4.1 TIME-COMPLEXITY AND THE STRUCTURE OF PROGRAMS

In this section, we will explore the relation between Time-Complexity C_T and the structure of the ALGOL-like subset we have chosen in the previous chapter. Our interest is in worst case analysis, i.e., we are interested in upper bounds. Complexity is given in terms of input variable $\bar{x}$, so for a block B with input $\bar{x}$ we shall denote its complexity by $C_T(B)(\bar{x})$.

4.1.1 Assignment Statements

C_T for assignment statements entirely arise from the time complexity of expression evaluation. Since we have not considered functions as a part of the subset we can expect the value of C_T to be a constant. We will express this as:

$$C_T(\text{assignment statement})(\bar{x}) = O(1) \text{ where } \bar{x} \text{ is an input.}$$

4.1.2 Sequential Composition

The formula for C_T for sequential composition $B_1; B_2$ is straightforward.

$$C_T(B_1;\ B_2)(\bar{x}) = C_T(B_1)(\bar{x}) + C_T(B_2)(\bar{y})$$

where

$$\bar{y} = f_{B_1}(\bar{x}).$$

4.1.3 Conditional Composition

For the two cases, we have considered in the language, the assignment of C_T is straightforward for this form of composition also.

(i) C_T (<u>If</u> $p(\bar{x})$ <u>then</u> B_1) $(\bar{x}) = C_T(B_1)(\bar{x})$

(ii) C_T (<u>If</u> $p(\bar{x})$ <u>then</u> B_1 <u>else</u> $B_2)(\bar{x}) = O(C_T(B_1)(\bar{x})) + O(C_T(B_2)(\bar{x}))$

4.1.4 <u>Iterative Composition</u>

$$B : \text{while } p(\bar{x}) \text{ do } B_1.$$

This is the first case where the use of The Fixpoint Theory comes in. As can be noted the number of times the iteration is performed is determined by the successive values of the variable being tested in the controlling predicate $p(\bar{x})$, hereafter called the controlling variable (recall a restriction placed on controlling predicates in Chapter 3).

Thus if x_j is a control variable, where $\bar{x} = (x_1, x_2, \ldots, x_j, \ldots, x_n) \in$ and $x_j \in D_j$, then we are interested in finding the time complexity of

$$B : \underline{\text{while}}\ p(x_j)\ \underline{\text{do}}\ B_1.$$

The function that gives the next value of the control variable is denoted by g in the rest of this chapter.

4.1.4.1 <u>A Method</u>: For the iterative composition,

$$B : \underline{\text{while}}\ p(x_j)\ \underline{\text{do}}\ B_1$$

the equivalent recursive program is

$$F(\bar{x}) \Leftarrow T_1(F)(\bar{x})$$

where

$$T_1(F)(\bar{x}) = \underline{\text{If}}\ p(x_j)\ \underline{\text{then}}\ F(f_{B_1}(\bar{x}))\ \underline{\text{else}}\ \bar{x}.$$

$$\therefore C_T(B)(\bar{x}) = C_T(F_{T_1})(\bar{x}).$$

Partition B_1 in to two parts B_1' and B_1'' such that B_1' captures only that part of B_1 which deals with the computation of the next value of controlling variable x_j and B_1'' captures that part of B_1 which pertains to the remaining body of the loop.

Then

$$C_T(F_{T_1})(\bar{x}) = C_T(G_T)(\bar{\bar{x}}) * C_T(B_1'')(\bar{x})$$

where

$$G(\bar{\bar{x}}) \Leftarrow \underline{\text{If}}\ \ p(x_j)\ \underline{\text{then}}\ G(f_{B_1'}(\bar{\bar{x}}))\ \underline{\text{else}}\ \bar{\bar{x}}.$$

and $\bar{\bar{x}}$ has as components x_j and all other variables (components) in $\bar{x}$ on which the next value of x_j is dependent.

$\therefore\ \ T(G)(\bar{\bar{x}}) = \underline{\text{If}}\ p(x_j)\ \underline{\text{then}}\ G(f_{B_1'}(\bar{\bar{x}}))\ \underline{\text{else}}\ \bar{\bar{x}}$.

For any $\bar{\bar{x}}$, the maximum number of conditions (predicate) to be checked to compute $T^i(w)(\bar{\bar{x}})$ is i. So $C_T(T^i(w)) = O(i)$. This C_T is in terms of integer i which corresponds to the superscript of T. It is indirectly concerned with input $\bar{\bar{x}}$.

For any given $\bar{\bar{x}}$, i.e., x_j, there exists $i \geqslant 1$ such that $T^i(w)(\bar{\bar{x}})$ gives the value of $G_T(\bar{\bar{x}})$. For this $\bar{\bar{x}}$, $T^k(\bar{\bar{x}})$ is defined for every $k \geqslant i$ but the same number of computations, i, are needed. (If $G_T(\bar{\bar{x}})$ is defined).

$\therefore\ C_T(G_T)(\bar{\bar{x}}) = O(C_T(T^i(w)))$; where $C_T(T^i(w))$ is related with input x_j.

To find the complexity of G_T in terms of x_j we have to relate i with x_j; say a relation $i = h(x_j)$.

To find h, one clue we have is the nature of change of x_j for each iteration. This can be arrived at from the relation between $T^i(w)$ and $T^{i+1}(w)$; $T^{i+1}(w) = T(T^i(w))$.

In B_1', x_j is changed to $g(x_j)$. So the value of x_j which requires maximum number of iterations, i.e., i iterations, computed over $\bar{T}^i(w)$ is such that $p(g^{i-1}(x_j))$ holds (i.e., $p(g^k(x_j))$ holds for every $k \leq i-1$) but $p(g^i(x_j))$ does not hold.

In general, there may be more than one value x_j satisfying this condition. Let us denote the set of these values by S^i, i.e.,

$$S^i = \{ x_j \in D_j \ / \ p(g^{i-1}(x_j)) \text{ and not } p(g^i(x_j)) \}$$

So

$$D_j = \bigcup_{i=1}^{\infty} S^i$$

Intuitively, S^i is that subset of D_j such that for its members i iterations are necessiated in the computation of the function represented by the while loop.

Now we pick an element e^i of S^i such that

(i) $g(e^i) = e^{i-1}$ for every $i \geqslant 2$

and (ii) there exists a function $r : N \longrightarrow D_j$

(N is a set of positive integers) for which

$r(i) = e^i$ and r is one-to-one.

That is $g(r(i)) = r(i-1)$. This leads us to the solution $h = r^{-1}$.

$$\therefore \ C_T(G_T)(\bar{\bar{x}}) = O(r^{-1}(x_j)).$$

4.1.5 <u>Procedure</u>

For the two cases, we considered in the language, the assignment of C_T is straightforward.

(i) Non-recursive Procedure:

<u>Procedure</u> P; B

Here complexity can be given by:

$$C_T(p)(\bar{x}) = C_T(B)(\bar{x}).$$

(ii) Recursive Procedure:

<u>Procedure</u> P; B [P] .

Complexity formula can be given as:

$$C_T(P)(\bar{x}) = C_T(f_{\text{call } P})(\bar{x})$$

where $f_{\text{call } P}$ is the least fixpoint of the recursive program $F(\bar{x}) \Leftarrow f_{B[P]}(\bar{x})$, where occurrences of <u>call P</u> will be replaced by F in the semantic definition $f_{B[P]}$.

So long as constraints on the variable controlling recursion are similar to those imposed on iterative compositions, viz., while loops, a technique similar to the one proposed in 4.1.4.1 can be used.

4.2 <u>EXAMPLES</u>

<u>Example 1</u>

Consider the program P to find the factorial x ! for given non-negative integer x ! .

```
P: y := 1;
   while  x > 0 do
               begin y:= y*x;
                     x:= x-1
               end;
```

$$C_T(P)(x,y) = C_T(y := 1)(x,y) + C_T(B)(x,y)$$

where B is the while loop.

$$\therefore C_T(B)(x,y) = C_T(Y := y*x)(x,y) * C_T(B_1)(x)$$

where $C_T(B_1)(x) = C_T(F_T)(x)$ and F_T is the least fixpoint of $F(x) \Longleftarrow T(F)(x)$;

where $T(F)(x) =$ <u>If</u> $x > 0$ <u>then</u> $F(x-1)$ <u>else</u> x.

$\therefore g(x) = x-1$.

Now $T^0(w) = w$

$T^1(w)(x) =$ <u>If</u> $x > 0$ <u>then</u> $T^0(w)(x-1)$ <u>else</u> x

$=$ <u>If</u> $x > 0$ <u>then</u> w <u>else</u> x

$T^2(w)(x) =$ <u>If</u> $x > 0$ <u>then</u> $T^1(w)(x-1)$ <u>else</u> x

$=$ <u>If</u> $x > 0$ <u>then</u> (<u>If</u> $(x-1) > 0$ <u>then</u> w <u>else</u> $(x-1)$)

<u>else</u> x.

$\therefore S^1 = \{0\}, \quad S_2 = \{1\}$.

S^1 has only one element so no other choice of e^1. $\therefore e^1 = 0$

Also $g(e^2) = e^1 \Longrightarrow e^2 = 0 + 1 = 1$ which is an element of S^2.

From the relation $g(e^i) = e^{i-1}$,

we have $e^i - 1 = e^{i-1}$.

$e^1 = 0 \quad \therefore$ By induction, we get $e^i = i - 1$.

$\therefore$ Take $r(i) = i-1$ for every $i \geq 1$.

$\therefore r$ is one-to-one.

$\therefore C_T(F_T)(x) = O(r^{-1}(x))$

$= O(x+1)$

$= O(x)$.

Thus

$C_T(P)(x,y) = O(1) + O(1) * O(x)$

$= O(1) + O(x)$

$= O(x)$.

<u>Example 2</u>

Consider the block B where m and n are non-negative integers.

```
B : while n > 1 do begin  m := n;
                          n := n-1;
                    while  m > 1  do m := m div 2
                    end;
```

Here $C_T(B)(n,m) = C_T(B_1')\ (n) * C_T(B_1'')(n,m)$

where

```
B'_1 : while n>1 do n := n-1
B''_1 : m := n; while m > 1 do m := m div 2
```

$\therefore \quad C_T(B_1'')(n,m) = O(1) + C_T(G_T)(n).$

where $G(m) \Leftarrow$ If $m > 1$ then G (m div 2) else m.

Now from Example 1, $C_T(B_1')(n) = O(n)$.

To find $C_T(G)(m)$:

$T(G)(m) =$ If $m > 1$ then G (m div 2) else m

$\therefore \quad g(m) = m$ div 2.

Now $T^1(w)(m) =$ If $m > 1$ then w else m

$\therefore \ S^1 = \{0,1\}$

$T^2(w)(m) =$ If $m > 1$ then $T^1(w)$ (m div 2) else m

= If $m > 1$ then (If (m div 2) > 1 then w else (m div 2))

else m.

$\therefore \ S^2 = \{2\}$.

If we take $e^1 = 0$ then $g(e^2) = e^1$ implies $e^2 = 0$. But $0 \notin S^2$; also $r(1) = 0$, $r(2) = 0$

$\therefore$ r does not become one-to-one.

$\therefore$ we take $e^1 = 1$. Also $e^1 = 1$ $\therefore$ $e^2 = 2$ which is in s^2.

$\therefore$ By induction, from $g(e^i) = e^{i-1}$, we get unique $e^i = 2^{i-1}$ For $\forall i>1$.

Note that $e^1 = 1 \in s^1$; $e^2 = 2 \in s^2$ and g is one-to-one; so e^i, got by $g(e^i) = e^{i-1}$ by induction for every $i > 1$, is in s^i.

Take $r(i) = 2^{i-1}$. $\therefore$ r is one-to-one.

$$\begin{aligned}\therefore\ C_T(G_T)(n) &= O(r^{-1}(n)) \\ &= O(\log_2 n + 1) \\ &= O(\log_2 n).\end{aligned}$$

Thus

$$\begin{aligned}C_T(B)(n,m) &= O(n) * (O(1) + O(\log_2 n)) \\ &= O(n) * O(\log_2 n) \\ &= O(n \log_2 n)\end{aligned}$$

Hereafter the base of log is taken as 2; if it is not mentioned.explici

Example 3

Consider the following program of Heapsort which rearranges n numbers A(1), A(2), ..., A(n) in increasing order.

```
Procedure HEAPSORT(A,n)
    begin Call HEAPIFY (A,n);
          i :=n;
          while i ≥ 2 do begin t := A(i); A(i) := A(1);
                                    A(1) := t;
                               Call ADJUST (A,1,i-1);
                                    i := i-1
                          end
    end.
```

```
Procedure HEAPIFY (A,n)
  //Readjusts the elements in A(1:n) to form a Heap.//
    begin i := n div 2;
          while i > 0 do begin Call ADJUST(A,i,n);
                               i := i-1
                         end
    end.

Procedure ADJUST (A,i,n)
  //The complete binary trees with roots A(2i) and A(2i+1) are combined
    with A(i) to form a single heap, 1 ≤ i ≤ n.//

    begin j := 2 * i; item := A(i);
          while j ≤ n do begin If j < n and A(j) < A(j+1) then j := j+1;
                               If item ≥ A(j) then exit
                                              else begin A(⌊j/2⌋) = A(j);
                                                         j := j *2
                                                   end
                         end
      A(⌊j/2⌋) := item
    end.
```

$\therefore$ $C_T(\text{HEAPSORT})(A,n) = C_T(\text{HEAPIFY})(A,n) + O(1) + C_T(B)(A,n)$

where B is the while loop in the procedure HEAPSORT. Breaking B into parts, we see that

$$C_T(B)(A,n) = C_T(B')(n) * (O(1) + C_T(\text{ADJUST})(A,1,n-1))$$

where

B' : while $i > 1$ do $i := i-1$.

$\therefore$ $C_T(B')(n) = O(n)$.

To find $C_T(\text{ADJUST})(A,i,n)$:

We note that in the while loop j is either changed to j+1 then j = j*2 or j = j*2 without becoming j+1. In the latter case more iterations are necessiated. Also we assume that item < A(j), for every j, so that as many as possible iterations are executed. This we do because we are finding worst case complexity.

$\therefore$ The while loop in ADJUST is equivalent to

<u>while</u> $j \leq n$ <u>do</u> $j := j*2$.

$\therefore C_T(\text{ADJUST})(A,i,n) = O(1) + C_T(G_T)(2i,n) + O(1)$.

where

$G(j,n) =$ <u>If</u> $j \leq n$ <u>then</u> $j = j*2$ <u>else</u> j.

Here $g(j) = j * 2$.

$T^1(w)(j,n) =$ <u>If</u> $j \leq n$ <u>then</u> w <u>else</u> j

$\therefore S^1 = \{j \,/\, j > n\}$.

$T^2(w)(j,n) =$ <u>If</u> $j \leq n$ <u>then</u> (<u>If</u> $(j*2) \leq n$ <u>then</u> w <u>else</u> $j*2$) <u>else</u> j.

$\therefore S^2 = \{j/j > n/2\}$.

Take $e^1 = n+1$ then $g(e^2) = e^1$ implies $e^2 = (n+1)/2$ which is in S^2. Also g is one-to-one. So we get unique e^i which is $(n+1)/2^{i-1}$. for $\forall i>1$.

$\therefore r(i) = \frac{n+1}{2^{i-1}}$ for every $i \geq 1$.

$$\therefore C_T(G_T)(j,n) = O(r^{-1}(j))$$
$$= O(\log_2(\tfrac{n+1}{j}) + 1)$$

$$\therefore C_T(G_T)(2i,n) = O(\log(\tfrac{n+1}{2i}) + 1)$$

$$\therefore C_T(\text{ADJUST})(A,i,n) = O(\log(\tfrac{n+1}{2i}) + 1)$$

$\therefore$ Now. $C_T(\text{HEAPIFY})(A,n) = C_T(i := n \text{ div } 2)(A,n)$
$+ C_T(H_T)(n \text{ div } 2) * C_T(\text{ADJUST})(A,n \text{ div } 2,n)$

where

$H(i) =$ <u>If</u> $i > 0$ <u>then</u> $H(i-1)$ <u>else</u> i.

$\therefore C_T(H_T)(i) = O(i)$.

$\therefore$ C_T(HEAPIFY)(A,n)

$$= O(1) + O(n \text{ div } 2) * O(\log(\frac{n+1}{n}) + 1)$$

$$= O(1) + O(n) * O(1) \qquad (\because n \geqslant 1)$$

$$= O(n)$$

$\therefore$ C_T(HEAPSORT)(A,n)

$$= O(n) + O(1) + O(n) * C_T(\text{ADJUST})(A,1,n-1)$$

$$= O(n) + O(1) + O(n) * O\ (\log\frac{(n-1)+1}{2})$$

$$= O(n) + O(1) + O(n) * O(\log\frac{n}{2})$$

$$= O(n) + O(1) + O(n \log n)$$

$$= O(n \log n)$$

Example 4

We will now see how to apply the fixpoint theory to find the complexity of simple programs having data structure other than integers. Our data domain should be ordered with a suitable ordering relation.

For most of data domains, if we impose an order relation on the given domain such that there exists a function L from this domain to the set of integers such that it ~~[illegible]~~ preserves order relation imposed on the domain, then we may apply the fixpoint theory.

In that case, in place of data elements $\bar{x}$ of a given domain, we can substitute the corresponding value $L(\bar{x})$ and can find the complexity by the fixpoint theory. As an illustration:

Let S be the set of all strings over some alphabet.

If we take function L as length of the string, denoted by $l(x)$, then we can define an order relation $\sqsubseteq$ on S by : for $x,y \in S$, $x \sqsubseteq y$ if and only if length $(x) \leq$ length (y).

So l ~~is one-to-one and~~ preserves the order relation $\sqsubseteq$.

Consider the following recursive program which appends string x to y.

$F(x,y) \Leftarrow$ <u>If</u> $x = \Lambda$ <u>then</u> y <u>else</u> $head(x) * F(tail(x),y)$.

where Λ denotes the null string.

Now $l(\Lambda) = 0$ and for $x \neq \Lambda$, $l(head(x)) = 1$ and $l(tail(x)) = l(x)-1$.

So equivalent recursive program is (in view of time complexity)

$G\ (l(x),l(y)) \Leftarrow$ <u>If</u> $l(x) > 0$ <u>then</u> $G(l(x)-1,\ l(y))$ <u>else</u> $l(y)$.

$$\therefore\ C_T(F_T)(x,y) = C_T(G_T)(l(x),\ l(y))$$
$$= O(l(x)).$$

**

CHAPTER 5

CONCLUSION

We have briefly presented basic ideas in The Fixpoint Theory of computation. Then, we tried to relate arguments concerning complexity to the structural specification of a rudimentary language. From the latter effort, in retrospect, it seems that the simplest way to treat iterative and recursive programs is to synthesize two programs from them, one which pertains to the gross control structure of the program and the other that encapsulates the work that is repeatedly carried out. Effecting such a synthesis is a non-trivial task, comparable in nature to the discovery of an invariant assertion in the Inductive Assertion Method of proving properties of programs (Manna 74). Once such a break-down of the original program is achieved, we can then study the two subparts separately: from the control structure, we try to arrive at a counting function that allows the assessment of number of iterations; the repeated action in the original program is recursively studied with respect to its structure.

With regard to the control structure study, it seems as though the most important aspect here is the ordering on the data domain that the program manipulates. The discovery of this ordering is a challenge comparable to the discovery of a well-formed set and its relationship with a program in proofs concerning termination of program.

Our application of these ideas to programs has been restricted to very simple programs wherein the clarity of analysis has been easily realised. Extending it to complex programs, i.e., with complex data domains, orderings and controlling predicates, is a taste beyond the scope of this work.

One satisfaction we have is the separation of concerns that arises from the study of complexity of iterative programs: the control structure has been abstracted out of program and is then 'composed' with the body of the loop. This separation of concerns is consistent with Backus'(78) critique of programming languages and his plea for functional languages.

The only claim we can make is with regard to the idea that we should formalize the structure of proofs of complexity by dealing with this problem as the level of programming language specification.

As is only too clear, much work needs to be done before a systematic structure for proofs of complexity can be given formally.

REFERENCES

Aho, A., Hopcroft, J.E., and Ullman, J.D., (1974); "The Design and Analysis of Computer Algorithms", Addison -Wesely Publishing Co., 1974.

Backus, John (1978): "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", CACM, August 1978.

Birkhoff, G., (1940); "Lattice Theory", American Mathematical Society Colloquium Publications, Volume XXV, 1940.

DeBakker, J.W., (1975): "Foundations of Computer Science", Mathematical Centre Tracts 63, Amsterdam, 1975.

DeBakker, J.W., (1973): "Recursive Procedures", Mathematical Centre Tracts 24, Amsterdam, 1973.

Loeckx, Jacques, (1974): "The Fixpoint Theorem and The Principle of Computational Induction", Lecture given at the Advanced Course on the Semantics of Programming Languages, Saarbrucken, 1974.

Manna, Zohar (1974): "Mathematical Theory of Computation", McGraw-Hill Kogakusha Ltd., 1974.

Manna, Zohar and Vuillemin, J., (1972): "Fixpoint Approach to the Theory of Computation", CACM, July 1972.

McCarthy, John, (1960): "Recursive Functions of Symbolic Expressions and Their Computation by Machine : Part I," CACM, April 1960.

Reynolds, J.C., (1972): "Notes on a Lattice-Theoretic Approach to The Theory of Computation", Systems and Information Science, Syracuse Univ., New York, October 1972.

Scott, Dana (1970): "Outline of a Mathematical Theory of Computation", Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems (1970), pp. 169-176.

Scott, Dana (1971): "Lattice Theory, Data Types and Semantics", New York University Symposia in Areas of Current Interest in Computer Science, (Randall Rustin, ed.), 1971.

Scott, Dana (1972): "Data Types as Lattices", Lectures by Data Scott for the Advanced Course on Programming Languages and Data Structures, Organized by the Mathematical Centre, Amsterdam, June 1972.